ABSTRACT

Dependencies between program elements need to be modeled from different perspectives reflecting architectural, design, and implementation level decisions. To avoid erosion of the intended structure of the code, it is necessary to explicitly codify these different perspectives on the permitted dependencies and to detect violations continuously and incrementally as software evolves.

We propose an approach that uses declarative queries to group source elements—across programming language module boundaries—into overlapping ensembles. The dependencies between these ensembles are also specified as logic queries. The approach has been integrated into the incremental build process of Eclipse to ensure continuous checking, using an engine for tabled and incremental evaluation of logic queries. Our evaluation shows that our approach is fast enough for day-to-day use along the incremental build process of modern IDEs.

Categories and Subject Descriptors:
D.2.2 [Design Tools and Techniques]: Programmer workbench;
D.2.8 [Design]: Representation

General Terms: Design

Keywords: continuous checking, controlling program dependencies, static analysis, Datalog

1. INTRODUCTION

Constraints on structural dependencies between elements of a software need to be expressed at different levels of abstraction. There are architectural level constraints, e.g., on dependencies between layers in a layered architecture. Further, there are design level constraints, e.g., stating that only factory classes can access constructors of product classes when the factory pattern [12] is employed. Finally, there are implementation level constraints, e.g., stating that the fields of a certain class can only be accessed via getter and setter methods of the same class.

Expressing constraints at different levels of abstraction implies that arbitrary groups of source elements can be defined and related to each other. The example architectural level constraint above relates layers of an application implemented by groups of classes, fields, and methods. The example design level constraint relates groups of classes to groups of methods in other classes. The example implementation level constraint relates groups of fields to groups of methods within the same class. We use the term ensemble to denote any logical grouping of code elements affected by a structural dependency constraint to be expressed. That is, classes, methods, and fields participating in the implementation of a layer may constitute an ensemble; the set of constructors of classes playing the product role in an instantiation of the factory pattern or the set of setter and getter methods of a certain class are further examples of ensembles.

Module-centric visibility mechanisms supported by programming languages, e.g., visibility modifiers of Java, are insufficient for expressing constraints on structural dependencies at different levels of abstractions, because they lack two properties, that ensembles possess:

First, ensembles can be defined orthogonal to the module system of the implementation language, e.g., the ensemble that groups all constructors of classes playing the product role in a factory pattern instantiation cuts across class boundaries. Second, ensembles can share members. For example, a class may be part of a layer ensemble and of the product ensemble in an implementation of the factory pattern. The participation in different ensembles imposes different constraints on the allowed dependencies that are in effect simultaneously.

This paper presents an approach to express constraints on structural dependencies between arbitrary ensembles, which are continuously enforced as the software evolves. Similar to the enforcement of visibility constraints expressed by Java visibility modifiers, which happens continuously as part of the regular build process, constraints on dependencies between ensembles need to be continuously enforced as the software evolves. This ensures that the implementation of a software system always conforms to its intended dependency structure, which is crucial for code comprehension, reuse, and maintainability. Continuous checking enables developers to fix issues as soon as they occur and is a prerequisite to prevent design erosion [37], and architectural erosion [29].

The contributions of this paper are as follows:

- A domain-specific language embedded in Datalog [6] is proposed for defining ensembles and for expressing constrains on their dependencies. This language also
2. SPECIFYING DEPENDENCIES

This section presents the notations for specifying ensembles and constraints on their structural dependencies. We illustrate the notations by constraining the structural dependencies of an example software system - the Bytecode Analysis Toolkit (BAT)\(^1\), a library for analyzing Java bytecode.

### 2.1 Logic-Based Core Specification Language

The domain-specific logic-based language for expressing ensembles and constraints on their dependencies, called LogEn (for Logical Ensembles), is embedded into Datalog\(^2\) [6].

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>package bat.type;</code></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><code>/** belongs to ensemble: TypesFlyweightFactory */</code></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><code>class TypeFactory {</code></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><code>    ObjectType getObjectType(String fqn) {</code></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td><code>    Object o = pool.get(fqn);</code></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td><code>    if (o == null) {</code></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td><code>        o = new ObjectType(fqn);</code></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td><code>        pool.put(fqn, o);</code></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td><code>    return o;</code></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td><code>}</code></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td><code>class ObjectType extends ReferenceType implements IType {</code></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td><code>    String fqn;</code></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td><code>    /** belongs to ensemble: TypesFlyweightCreation */</code></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td><code>    ObjectType(String fqn) {</code></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td><code>    </code></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td><code>    }</code></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Listing 1: Example BAT classes**

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>type(t1, 'bat.type.ObjectType')</code></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><code>type(t2, 'java.lang.Object')</code></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><code>type(t3, 'java.lang.String')</code></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><code>type(t4, 'bat.type.IType')</code></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td><code>superclass(t1, t2)</code></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td><code>interface(t1, t4)</code></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td><code>method(m1, t1, '&lt;init&gt;', [t3], t1)</code></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td><code>field(f1, t1, 'fqn', t3)</code></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td><code>type(t5, 'bat.type.FactoryType')</code></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td><code>method(m2, t5, 'getObjectType', [t3], t1)</code></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td><code>uses(m2, m1)</code></td>
<td></td>
</tr>
</tbody>
</table>

**Listing 2: Representation of Java code**

Datalog is a subset of Prolog that ensures termination and allows tabling of the query results without distorting the semantics of queries. This is a crucial prerequisite for the incrementalisation of queries. Because of its good runtime efficiency compared to full Prolog, Datalog is argued to be a good choice for program queries [13]. We use the XSB\(^3\)-engine for our implementation and therefore use XSB syntax and terminology in the following.

### Representation of Java Programs

Source elements of Java programs are represented as relations defined by a database of Datalog facts. The following Java elements are represented: class, interface, method and field declarations, method calls, field accesses, annotations, exception declarations and exception handling constructs. The representation has been designed to balance the need to conserve as much information about the structure of the Java source as is needed to enable the specification of all types of constraints related to compile time dependencies with the need to keep the representation compact enough to allow speedy conversion and to save memory. For this reason, intra-method control structures such as loops and switches are not represented, since they neither serve as means to group source elements into ensembles nor take part in dependencies we want to control.

To illustrate the proposed Datalog-based representation of Java programs, consider the classes from the BAT project in Listing 1 showing `TypeFactory`, a factory class that produces flyweights of `ObjectType` objects; Listing 2 shows the encoding of the class `ObjectType`.

Each source element is associated with an unique id. Type declarations are encoded using `type/2`\(^4\) (see, e.g., Line 1 in Listing 2), where the first argument (or, parameter) is the id (identifier) of a source element and the second parameter its type name. Superclass relationships are encoded using `superclass/2` (Line 5); the ids of the base type and of the super type are given as arguments. Inherited interfaces are encoded by `interface/2` (Line 6); the first argument is the id of a type and the second argument the id of the inherited interface. Annotations, thrown exceptions and visibility modifiers are encoded in a similar way. Method declarations are encoded by `method/5` (Line 7) with the following arguments: the method id, the id of the defining class, the method’s name, the list of ids of the parameter types, and the return type’s id. Field declarations are encoded using `field/4` (Line 8); the arguments encode the field id, the id of

\(^{1}\)http://www.st.informatik.tu-darmstadt.de/BAT
\(^{2}\)extended with arithmetic and negation using a closed world assumption
\(^{3}\)http://xsb.sf.net
\(^{4}\)The notation /2 denotes a relation with arity 2.
the defining class, the field name, and the type of the field. Further, some rules are available, that build upon these relations. For example, `inherits/2` relates the class given as the first argument to all its direct or indirect supertypes.

The uses relation between source elements is represented by `uses/2`, whose first argument is the id of a source element that uses the source element specified as the second argument. In Java, a (syntactic) use exists between two source elements SA and SB, if SB is used in SA. If SA and SB are method declarations, a use is e.g., a call of SB in SA. If SB is an exception type, the declaration of SA to throw SB, the creation of an instance of SB in SA, or a catch statement for exceptions of type SB are all uses of type SB are all uses of SB in SA. For an example, see Listing 2, Line 11, which states that the definition of the method `getErasedType` in class `TypesFactory` (Line 5 in Listing 1 and Line 10 in Listing 2) uses the constructor of `ObjectType` (Line 17 in Listing 1 and Line 7 in Listing 2).

### Constraints on Dependencies between Ensembles

Source elements that belong to an ensemble are identified using the `part_of/2` relation, where the first argument is a source element and the second parameter is the name of the ensemble. The rules defining this relation build upon the source code representation and predefined relations, some of which were discussed in the previous sub-section. For illustration, three ensembles are defined in Listing 3. The `Types` ensemble (Line 1) comprises all source elements in any package starting with `bat.type`. The `TypesFlyweightFactory` ensemble matches the whole factory class (Line 5) and the `TypesFlyweightCreation` ensemble (Line 9) matches all constructors of all classes that inherit from `IType`.

Dependencies between ensembles are derived from the uses between source elements, as schematically shown in Figure 1. Ensemble A depends on ensemble B, if there is a source element SA that is part of A that uses source element SB that is part of ensemble B.

To specify constraints on dependencies between ensembles `violations(S, T, 'constants')` is used, where `constants` is the name of the constraint; the relation binds the source (S) and target (T) code elements of uses relations that violate constraint `‘constName’`. To formulate rules defining this relation, only `part_of/2`, conjunction, disjunction, and tabled negation [32] are used in our language. The specification as a whole is valid if `violations/3` does not do hold for any uses and constraints. The examples throughout the paper and our experiments presented in Section 3 show that this is expressive enough to formulate a wide range of constraints.

For illustration, consider violations(S, T, 'TypesFlyweight') in Listing 4. It uses the ensembles `TypesFlyweightCreation` and `TypesFlyweightFactory` defined in Listing 3 to specify that only the class `TypesFactory` is allowed to create instances of `ObjectType` and its subtypes. The rule returns all pairs (S, T), such that `uses(S, T)` is true, T belongs to `TypesFlyweightCreation`, and S does not belong to `TypesFlyweightFactory`. For an example of a violation of the constraint expressed by violations(S, T, 'TypesFlyweight') consider the method `getErasedType` in Listing 4 and assume that it has the id m3 in the corresponding logic representation. The call in Line 10 uses the constructor of `ObjectType` (with the id m1 in Listing 2), i.e., `uses(m3, m1)` is true. This violates the 'TypesFlyweight' constraint, because `part_of(m1, 'TypesFlyweightCreation')` is true and `part_of(m3, 'TypesFlyweightFactory')` is false.

### Templates

In LogEn, templates define recurring patterns of constraints on structural dependencies. A template is a violations/3 rule, whose third argument is not a ground term (i.e., constant), but a variable or a term containing variables. Using variables in the argument representing the ensemble allows to formulate a rule that matches all ensembles that can be unified with the argument term. For illustration, consider the template `violations(S, T, flyweight(Instance))` in Listing 5. This template specifies a con-
straint on structural dependencies between participants of the flyweight pattern, stating that only the factory is allowed to create flyweights. This specification uses the variable Instance in the third argument to abstract over any particular instantiation of the flyweight pattern. violations(S, T, 'TypesFlyweight') in Listing 4 specifies the same constraint for only one particular instantiation of the flyweight pattern, whose participants are the classes in the TypeFactory package and the class ObjectType and its subclasses.

To instantiate the template violations(S, T, flyweight(Instance)) for a particular instance of the flyweight pattern, the Instance variable needs to be bound to the same name for all ensembles participating in the template instantiation.

For example, to complete the template for the particular instantiation of the flyweight pattern in BAT presented in Listing 1 one would define the TypesFlyweightCreation and TypesFlyweightFactory ensembles as in Listing 3, but replace the constants 'TypesFlyweightCreation' and 'TypesFlyweightFactory' with flyweightCreation(types) and flyweightFactory'(types) respectively.

Listing 5 in Lines 5 to 8 shows a template defining a constraint on structural dependencies imposed by layered architectures [5]. This template specifies that the ensemble representing the implementation of a layer N is allowed to depend on any source element in implementations of layers M. This access restriction in the layered architecture is expressed by N >= M; the variable Instance abstracts over any specific instance of the layered architectural pattern. To use this template in a concrete context, e.g., to specify the layered architecture of BAT, which consists of three layers, the following steps need to be taken: (a) the Instance variable needs to be bound to a concrete name, e.g., bat, and (b) the extent of the resulting ensembles, layer(0,bat), layer(1,bat), and layer(2,bat), need to be defined by specifying respective part_of/2 queries.

2.2 Visual Dependency Specification

This section introduces VisEn (for Visual Ensembles), a visual language for specifying constraints on dependencies between high-level building blocks of an application; graphical notations are supposed to be more suitable for this kind of structural dependencies [19]. VisEn is mapped to LogEn specifications, as discussed later in this section.

Graphical Notation

VisEn is introduced by using it to express high-level constraints on structural dependencies between elements of our example system. To start with, Fig. 2 shows a VisEn specification of the high-level building blocks of the BAT frame-work and their allowed structural dependencies. Boxes denote ensembles and arrows denote allowed dependencies. The main building blocks of BAT are Core and IO; the other three ensembles in Fig. 2 represent functionality to generate different code representations. The specification states that Core must not depend on any other ensemble; Core does not have any outgoing dependencies. IO may only depend on Core, while the other ensembles may depend on both Core and IO.

Fig. 3 shows a more detailed view of Core and IO and illustrates how ensembles can be nested into each other. The nesting structure has two implications on allowed dependencies. First, an element of an ensemble E may implicitly depend on any element that directly belongs to E or to any of E’s enclosing ensembles (e.g. source elements in the Types ensemble may access source elements in Core). All other dependencies must be stated explicitly. E.g., elements of Writer Impl on the left-hand side of Fig. 3 may implicitly depend on any element el of IO, provided that el does not belong to a sibling sub-ensemble of Writer Impl such as Reader Impl; to enable elements of Writer Impl to depend on elements of JVM Constants, the explicit dependency from Writer Impl to JVM Constants is needed. Second, dependencies between sub-ensembles nested in two different enclosing ensembles E1 and E2 are allowed only if E1 and E2 depend on each other. This enables reasoning about ensembles in their folded state (as black boxes).

However, an explicit dependency between E1 and E2 does not automatically propagate to their sub-ensembles; for this purpose so-called in- and out-ports are provided. An out-port is denoted by a semi-circle attached outwards to the boxes (cf. Fig. 3) and is used to specify outgoing dependencies. An in-port is denoted by a semi-circle attached inwards to the boxes and collects incoming dependencies. The connection between the out-port on the right-hand side of IO to the in-port of Core specifies that IO as a whole depends on Core.

Sub-ensembles of an ensemble E use E’s out-ports to specify external dependencies, i.e., on ensembles outside E. The inner structure of IO reveals that JVM constants is not allowed to depend on Core, since it is not connected to IO’s corresponding out-port. An ensemble may have multiple out-ports to enable fine-granular modeling of external dependencies of its sub-ensembles. E.g., IO has two out-ports to express that only Reader may depend on the ensemble jakarta.commons.io, while ReaderImpl and WriterImpl may depend on Core. To avoid visual clutter, out-ports are only shown if there is more than one, or to specify external dependencies of a sub-ensemble; e.g., the single out-port of Core is shown in Fig. 3 to specify that a dependency between Data Structure and Javolution is allowed.

An ensemble E1 that is connected to an ensemble E2 may access source elements that directly belong to E2. However, E1 may depend only on sub-ensembles of E2 to which E2’s in-port is connected. Each ensemble has exactly one in-port. If an ensemble E had more than one in-port, an ensemble E1 depending on E would need to know about E’s internal structure to decide which in-port to use, making black-box reasoning impossible. In-ports are only shown if incoming dependencies need to be forwarded to nested ensembles. For illustration, consider Core and IO in Fig. 3. Core’s in-port

5The plus sign denotes ensembles with folded inner structure
has connections to sub-ensembles Types, Bytecode Representation, and Quadruples Representation. As a result, ensembles that depend on Core (IO and BAT2TXT) may depend on elements of these sub-ensembles, in addition to elements that directly belong to Core. IO’s in-port has no explicit connections, hence is not shown.

VisEn distinguishes between open, restricted, and internal ports. By default, a port drawn with a solid line is restricted; a hidden port is also restricted. Open ports are drawn as semi-circles with dashed lines and do not limit dependencies from or to an ensemble. An internal port is denoted by a dashed circle and grants the elements of an ensemble access to sub-ensembles connected to it.

Fig. 4 shows a visual specification of the constraint that only respective flyweight factories are allowed to create flyweights of object types in BAT (cf. the 'TypesFlyweight' constraints in Listing 4). This specification uses ensembles with open and restricted ports. The restricted in-port of TypesFlyweightCreation states that access to elements of TypesFlyweightCreation is exclusively granted to elements in TypesFlyweightFactory. Due to the open out-port, the specification does not constrain in any way dependencies of TypesFlyweightCreation on other source elements, e.g., helper classes.

For an example of using an internal port in a visual specification consider Core in Fig. 3. Its inner port enables Core’s elements to use elements of Data Structures; this cannot be expressed by an open in-port of Data Structures, because this would render the latter accessible for all ensembles outside Core that connect to Core’s in-port.

From VisEn to LogEn Specifications

This section discusses, how VisEn notations are mapped to LogEn. The LogEn predicates presented in this section constitute a sub-language of LogEn which can be directly used to express enclosing and dependency relationships between high-level ensembles. Yet, we believe that using the visual counterpart is more convenient and results in specifications that are easier to understand.

Visual nesting of ensemble boxes in VisEn is transcribed to isEnclosedIn/2 facts in LogEn. For example, as WriterImpl’s box is nested into IO’s box in Fig. 3, the fact isEnclosedIn('WriterImpl', 'IO') will be generated.

The relations outPort/2, inPort/2, and internalPort/2 relate the ids of out-ports, in-ports, and internal ports respectively with their ensembles. The ids of all open ports are represented by the unary relation isOpen/1. For example, inPort('Reader', P) ∧ isOpen(P) returns true and inPort('Javolution', P) ∧ isOpen(P) returns false.

Ports that are directly connected in visual specifications are encoded using p_connect/2.6 The transitive closure of p_connect/2 is defined by p_connect_trans/2. For example, the conjunction outPort('ReaderImpl', P1) ∧ inPort('Types', P2) ∧ p_connect_trans(P1, P2) is true, since a path connecting both ports exists in Fig. 3.

The rules in Listing 6 encode dependencies between ensembles. The rule depends(E1, E2) returns the ensembles on which a given ensemble depends or those depending on a given ensemble. It uses e_connect(E1, E2), which holds for two ensembles that are connected in a visual specification. For example, depends('Reader Impl', E) results in E = ['IO', 'JVM Constants', 'Types', 'Quadruples Representation', 'Bytecode Representation'] and depends(E, 'Data Structures') returns E = ['Core', 'Quadruples Representation', 'Bytecode Representation', 'Types']

The violates/3 rules in Listing 7 define what it means for pairs of source code elements (S,T) participating in the uses/2 relation to violate visually specified enclosing and outgoing and incoming dependency relations between ensembles. The enclosing constraint (Line 1) states that all sources of uses relations that are elements of a sub-ensemble also belong to the enclosing ensemble. This constraint is necessary, since given two ensemble specifications S1 and S2, it is not possible to statically decide whether S1 matches a subset of

---

6The in- and out-ports of the same ensemble are not considered connected.
elements matched by S2. The outgoing constraint (Line 6) is violated by any element of the uses(S, T) relation, where the source S is in an ensemble E that does not have an open out-port and the target T is not in E or in an ensemble on which E depends. The incoming constraint (Line 11) is violated by any element of the uses(S, T) relation, such that the target T is part of an ensemble E whose in-port is not open and the source S is part of neither E nor one of the ensembles depending on E.

2.3 Using Meta-Data to Define Ensembles

The primary means to associate source elements with ensembles is the part_of predicate of LogEn. Complementary to part_of/2, our approach also supports the use of meta-data as a means to associate source elements to ensembles. Definitions based on meta-data are appropriate for localized ensembles that participate in design and implementation level structural dependencies, as using metadata enables a definition of ensembles close to the affected source elements. For localized ensembles, such definitions tend to be less fragile and easier to comprehend and maintain when compared to logic-based specifications stored in separate artifacts. E.g., by labeling a class as a factory product, developers maintaining or using the class become aware of its role w.r.t. the factory pattern. Furthermore, if decisions about permitted design and implementation level dependencies change, it is possible to update the related specification(s) in place. The use of annotations also makes the definitions robust against refactorings, such as the renaming of affected source elements. This reduces the effort needed to maintain the ensemble structure, when compared with approaches, where the connection between the queries and the code is based on string matching.

To illustrate the use of meta-data recall the generic specification of dependency constraints related to the flyweight pattern (cf. Listing 5 in Sec. 2.1). Two generic ensembles were defined, flyweightCreation(Instance) and flyweightFactory(Instance); the template is instantiated by binding the variable Instance to a name that denotes the concrete instance of the pattern, e.g., types. In Listing 3, part_of/2 was used to associate source elements with ensembles in the dependency specification for the flyweight pattern. Using meta-data, the developer creates the types instance of the flyweightFactory(Instance) template by annotating TypeFactory with @FlyweightFactory("types") (see Lines 6–7 in Listing 8).

3. ENFORCING DEPENDENCIES

In this section, we describe how the approach is integrated into the incremental build process of an IDE to ensure continuous enforcement of constraints and to evaluate its performance.

3.1 IDE Integration

We have implemented the proposed approach for Java using the static analysis platform Magellan [9, 10]. Magellan is tightly integrated with the incremental build process of the Eclipse IDE and features an integration of the XSB\(^7\) engine. The integration into the incremental build process of Eclipse enables the implementation of queries over Java code that are continuously evaluated when code changes. Furthermore, the integration of the XSB engine enables that queries are evaluated in an incremental way, due to XSB’s support for automatic incrementalization [8, 33] of queries.

For ensembles that are on an architectural level, the definition of constraints on dependencies between ensembles is done in the visual language described in Section 2.2, which is mapped to the model as described there. Ensembles below the architectural level are defined using metadata attached to the source elements, namely annotations (as described in Section 2.3). The constraints for these ensembles are defined using the templates shown in Section 2.1.

All classes of a project are parsed and converted to the program model presented in Section 2.1 and added to the fact base. To get the set of violations, we evaluate violations/3 (described in Section 2.1) and present the returned uses, that violate the defined constraints, to the developer.

For incremental builds, the facts of removed classes are removed, and the representation of added classes is added to the fact base. After that, the ensemble definitions and ensemble dependency specifications that have changed are also updated. Due to the automatic incrementalization, the set

\(^7\)http://xsb.sourceforge.net
of violations is incrementally maintained and the updated set of violations is then presented to the developer.

### 3.2 Evaluation

In the following, we evaluate our approach for defining a software system’s structure as well as the full-build and incremental-build analysis times of it. We put, however, special emphasis on incremental build times as these builds are executed regularly by Eclipse whenever a document changes. Full builds are only executed on explicit user request or when a large part of a project changes, e.g. after a CVS checkout. In the latter case the user typically expects a longer build time and, hence, the additional analysis time is less critical. However, in case of incremental builds the result has to be immediately available as the developer directly wants to continue working.

For the evaluation, we used three systems of different sizes to be able to also reason about the scalability of the approach. The subject systems were: (a) the Bytecode Analysis Toolkit (BAT) described in Section 2.2, consisting of 849 classes in 22 packages totaling 120,000 LOC, (b) Jakarta regexp package³, consisting of 14 classes with 3,663 LOC, and (c) abc [1], consisting of 2,874 classes with 285,000 KLOC.

The results of the BAT experiment are presented first; the results for the other two experiments are presented subsequently relative to the results for BAT. BAT’s high-level architecture was defined as described in Section 2.2. The visual model of the high-level structure resulted in 17 ensembles with corresponding dependencies. The majority of these ensembles reflect the package structure, but some cut across the modular structure of Java, e.g. grouping constructors of a set of classes that create a different code representation to an ensemble. The use of annotations to model low(er) level structural dependencies resulted in another 36 ensembles. The annotations were used to model (a) the dependencies of BAT’s factories as discussed in section 2.2, (b) intra-class dependencies, e.g., that a field is only to be accessed by its getters and setters, and (c) other inter-class dependencies between elements that logically belong together, but which are spread over the project for technical reasons; for example, the constructor of the class which represents a method control-flow graph is only intended to be called by the createCFG() method of the class that manages a method’s code representation.

After specifying and checking for violations of the intended dependency structure, several violations were found and most of them could immediately be fixed by applying move method, and move class refactorings. However, severe violations were also found. For example, the class which manages a method’s bytecode — belonging to the Core ensemble — also offered a method to clone a method’s code to use it as a prototype for a new method. This method used functionality in the IO ensemble to write out the method as bytecode and then to directly reread the method. This clearly violated the intended structure where Core must not have any dependencies on IO. In short, after several years of development of BAT, involving the work of a large number of

³http://jakarta.apache.org/regexp/

<table>
<thead>
<tr>
<th>Change</th>
<th>Time</th>
<th>Uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100 ms</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>156 ms</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>173 ms</td>
<td>61</td>
</tr>
<tr>
<td>4</td>
<td>174 ms</td>
<td>60</td>
</tr>
<tr>
<td>5</td>
<td>167 ms</td>
<td>55</td>
</tr>
<tr>
<td>6</td>
<td>122 ms</td>
<td>60</td>
</tr>
<tr>
<td>7</td>
<td>1.285 ms</td>
<td>1.229</td>
</tr>
</tbody>
</table>

Table 1: performance measurements

(PhD) students, the main structure was still visible, but already showed structural erosion.

The performance was measured on a 2.3 GHz Core2Duo system with 2 GB RAM using Eclipse 3.3 and Java 1.5. We measured (a) the full build and (b) the incremental build process time. In the full build case, the whole analysis took 10 secs: 9 secs to generate the program representation (which comprises 51,278 facts) and 1 sec to check for violations.

Table 1 lists the results of our evaluation in case of incremental builds. The type of changes performed on BAT are described in the “Change” column; time is the overall time of creating the representation, updating the database and reevaluating the violations/3. The uses column shows how many uses are rechecked to indicate the complexity of the change.

Since our analysis is integrated with the Eclipse build system, the observable unit of change are entire artifacts (i.e. Java class files). The smallest changes (Lines 1 and 2 in Table 1), are the addition, resp. removal, of an empty class. Lines 3 and 4 show the effect of changing the inheritance hierarchy by letting the type IType inherit resp. no longer inherit the interface Serializable. Change 5 is the move of a class to a different ensemble. Change 6 represents the editing of a class — unrelated to the system’s structure — containing an annotation with a ensemble specification. Change 7 shows the effect of splitting an ensemble and moving 33 classes to a different package.

For the Jakarta regexp package, we specified four ensembles and three constraints. The full build analysis took 712 ms: 674 ms to build the program representation and 38 ms to check for violations. We performed changes comparable to the changes made for BAT and in all cases the incremental analysis times were approx. 1/10th of those in case of BAT (shown in Table 1).

As a large project we used abc [1], for which we identified more than 100 high level ensembles. The full build analysis took 61.8 secs, with 60.3 secs to construct the prolog representation and 1.5 secs to actually check for violations. Again, we performed the same type of changes as in case of BAT and in this case the analysis times are 3 to 4 times slower. As these results show, our approach does not scale linearly. Nevertheless, the performance is reasonably fast for projects with at least up to 250 KLOC.

The results indicate that checking the constraints on structural dependencies along with the incremental build process is fast enough to be used as part of the build process. Small changes usually take less than one second, and even significant refactorings take at most 2 seconds. The only changes that take longer are changes of the high-level structure, which are not relevant for the day-to-day work on code.
4. RELATED WORK

Formally, our approach can be seen as a Relational Partition Algebra (RPA) [11, 38]. Muskens [28] uses RPA to check consistency between multiple diagrams in a UML design or between a design and an implementation, e.g., to check if a method mentioned in a UML diagram, is absent in the implementation. Crocopat [2], which translates the relational expressions to BBDBs [4] uses them to detect design patterns and structural problems in the source code, whereas our approach checks for violation of structural dependency constraints. Postma [30] describes a method for module architecture verification using RPA, which targets high-level architectural rules only.

GraphLog [7] use a graphical notation that is translated to Datalog with negation to query software structure. GraphLog focuses on the visual query language that includes graphical representations of relations, although the mapping of the visual language to code is not in their focus.

CodeQuest [14] offers a fast, scalable code querying engine based on Datalog and intended for program understanding and refactoring support. The tool can also be used to check for coding style violations (e.g., public, non-private fields). CodeQuest is slower than XSB, but less memory intensive and therefor scales better for large projects on computers with limited memory. If memory usage turned out to be a problem, we could switch our backend to CodeQuest.

Languages specialized on software constraints like SCL [16] and its predecessor FCL [17] are more expressive in the constraints they can specify when compared to our approach, but they lack the modularization and incrementalization features of our approach. SCL uses a first order predicate logic based term language to reason about program source code. Our approach explicitly restricts the constraint language to be fast enough to evaluate and expressive enough to be useful for formulating architectural constraints. Furthermore, the authors of SCL admit, that the approach has a rather heavyweight notation. Our approach comes with a small, developer friendly visual and metadata interface.

Rigi [26] is a reverse engineering tool that generates a layered architecture with disjunct modules. It complements our tool nicely, as they focus on generating a structure where we focus on maintaining and validating the structure as part of an incremental build process.

The concern manipulation environment (CME) [15] provides a unified way to represent concerns across different types of software engineering artifacts. CME includes a concern exploration tool called ConMan that represents the concern space in a tree-based view. Concerns can be assigned elements either extensionally (by explicit reference), or intensionally via a fixed set of queries over relations, such as extends, implements, refersTo, or referencedBy. The focus of CME is the modeling of concerns in the context of aspect-oriented software development; a constraint definition language is not part of the proposal. Further, the querying capabilities of CME are restricted to a predefined set of predicates.

FEAT [31] is a tool to support software maintenance, using concern graphs. Developers bind source elements to concerns and build concern graphs semi-automatically as they traverse through code. Given a concern and a source element being navigated over, e.g., a method, the developer can use a fixed set of selectors, e.g., called methods, to add structurally related elements to the same concern. Concern graphs can also be made persistent. FEAT and our approach differ substantially in their focus. FEAT uses concerns and concern graphs for reverse engineering and comprehension purposes, to help the developer to localize and change the code implementing a certain concern. Hence, FEAT only documents existing relations between concerns, but does not impose constraints on these relations. On the contrary, we use ensembles as a means to express and enforce constraints (invariants) over their allowed dependencies. Another difference concerns support for incremental changes. In FEAT, incremental changes of source code leads to incremental changes of the model; however, the affected relations have to be recomputed. In our approach, incremental tabling [8] is used to recompute only the part of the affected relations that depends on changed facts.

The Intensional Views [23, 36, 22] approach uses logic meta-programming to codify programming patterns ranging from best practices (getter/setter) to design patterns and bad smells. The codified patterns are used to search for instances of patterns in a code base, to check for pattern violations, or even to generate code. There are important differences between our approach and intensional views and their derived proposals. The focus of intentional views is very broad covering the codification of arbitrary programming patterns which are matched against existing code structures. The focus of our approach is on (re)grouping existing program elements into ensembles which are organized in nested dependency (usage) graphs to be continuously enforced. This difference on focus has two consequences. On the one hand, we have not seen intentional views being used to partition program elements for the purpose of expressing dependencies across the boundaries of programming modules. On the other hand, to serve the broad focus, the program model of intentional views is much more detailed than ours and the query language has unrestricted expressiveness, including the use of quantifiers and unlimited recursion. Our logic-based language is domain-specific and solely serves the specific purposes of expressing dependencies that cut across built-in modules of the programming language. The differences on expressiveness have severe effects on the suitability of the languages to be used for writing queries to be executed along the incremental build process.

Software reflexion model [27] is an approach to prevent design erosion. The tool supports developers to check the conformance of a program to an architectural model specified by the architect. The mapping of parts of the program to disjunct modules is done declaratively by, e.g., specifying regular expressions that match the filenames. The developer specifies relations ("calls" or "communicates-with") between the identified modules. The relations between the parts of the program identified as modules and relations between the modules in the specified model are compared and the result is visualized as a graph. In our approach, the source elements can take part in more than one ensemble and can be defined at a fine level of granularity, e.g., class members can be specified to be part of a ensemble. Furthermore, our approach supports the incremental and continuous enforcement of constraints.

SonarJ\footnote{http://www.sonarj.org} is a commercial tool to model the architecture of a Java program as a set of disjunct, acyclically connected modules and to check for the conformance of the program to
the architecture. The granularity of SonarJ’s model is fixed and on package level. A simple kind of crosscutting structure can be expressed by dividing the software in a matrix-like structure, where the rows represent technical layers like view or controller and the columns represent business layers, like supplier or customer. This structure is then resolved into disjunctive modules which are named “row::column”. Dependencies and constraints are expressed using these modules. Our approach is more flexible, as it can model ensembles that crosscut the package structure.

Law Governed Architecture [24] is an approach to restrict architectural changes by enforcing laws on the architecture of evolving systems. The work includes ideas on how to anticipate certain degrees of change in the architecture of the system while maintaining the laws (comparing them to a base law, called constitution). Part of the realization of this concept is the Environment Darwin-E [25], which can check syntactic dependencies of Eiffel code using a Prolog-based static analysis. Our approach uses a specialized, restricted language to formulate the dependencies, thus enabling fast, incremental checking.

Hedgehog [3] uses its own Prolog like language called Spine to describe declarative constraints that define implementation restrictions for design patterns. The constraints describe structural properties of classes and method invocations. The goal of Hedgehog is to verify design patterns, in the sense that developers can check that design patterns are correctly implemented. However, since Hedgehog tries to automatically extract the implemented design patterns it does not detect all implemented design patterns and hence is not well suited to enforce the architecture. PEC [21] is a pattern enforcing compiler for Java. Using interfaces to identify the intended design pattern, the tool combines static testing, dynamic testing (unit testing), and code generation to verify, that the pattern is implemented according to specification. Pattern-lint [34] uses a combination of static and dynamic checking to confirm that the implementation of a system maintains its expected design models and rules. These approaches all target design patterns only. Our approach scales from modeling the top-level architecture down to intra-class design decisions.

Aspect-oriented languages provide modules for capturing crosscutting concerns. The proposed approach complements aspect oriented approaches because it supports the definition of crosscutting views over programs with the goal of expressing dependency constraints between these views. The goal of AOP is to capture crosscutting behaviors in a modular way into aspects. Dependencies between code elements involved in these aspects may equally be subject of constraints.

Shomrat [35] showed, that using AspectJ [18] to enforce architectural restrictions is not an ideal choice. Although design problems are cross-cutting, they often concern static events or structural properties that cannot be captured by existing pointcut languages. Using static analysis, as we do in our prototypical implementation seems better suited to ensure structural properties.

Lam and Rinard present a type system and analysis for the automatic extraction of design information [20], which is used to reverse engineer design information from existing systems. The type system is an extension to Java and uses type parameters as tokens to represent design elements. The tokens are placed on objects by parameterizing the object at creation time with the type parameter. A static analysis extracts design information from the program to construct models of the heap and object interactions. As the system does not allow the specification of architectural constraints, it is not possible to automatically detect unintended dependencies.

5. SUMMARY AND FUTURE WORK

We proposed and evaluated an approach that seamlessly integrates the specifying, visualizing and continuously checking of the structure of a software system. Central to the approach is the close integration into the incremental build process of an IDE. At the core of our approach is a domain specific language for specifying structural dependencies between source elements. The language is designed with continuous checking in mind and enables modeling of a software system’s structure at all levels of granularity; ranging from intra-class dependencies to architectural building blocks. A visual notation directly complements the approach by facilitating the comprehensive modeling of a system’s high level structure. By using annotations (meta-data), we enabled the refactoring resilient modeling of low(er) level structural dependencies.

In future work we will investigate extending our model with new predicates to make use of control- and data-flow information while still enabling incremental checking. This would support modeling of ensembles realising behavioral concerns. It is also interesting to exploit our tool for software understanding. Tools for extracting architectural views from code could be used to get a first overview of the dependency structure of the system. After that, our tool could be used to refine the model of this structure while continuously providing feedback about discrepancies between the specified structure and the implementation.

6. REFERENCES


